07D081M

07D081M

Project Number: CS-DJD-VFAC

Model Checking for Role-Based Access Control

A Major Qualifying Project Report:

submitted to the Faculty

of the

Worcester Polytechnic Institute

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Eric Griffel

Paul Sader

Date:

Approved:

Professor Daniel J. Dougherty

Professor William J. Martin

Abstract

An access-control policy is a formal description of the privileges that users have on resources in a system. This project is an implementation of a technique for modeling the combination of a system and a role-based access control policy for the purpose of static analysis through verification.

Contents

1 Access Control **3**
- 1.1 The Policy . . . 4
  - 1.1.1 Privilege Rules . . . 4
  - 1.1.2 Separation . . . 4
- 1.2 Verification . . . 5
- 1.3 Classic Access Control Paradigms . . . 7
  - 1.3.1 Discretionary Access Control . . . 7
  - 1.3.2 Mandatory Access Control . . . 7
- 1.4 Role-Based Access Control . . . 8
  - 1.4.1 Entities . . . 9
  - 1.4.2 Concepts & Principles . . . 10
  - 1.4.3 Datalog . . . 11

2 Related Work **13**
- 2.1 Change-Impact Analysis: Margrave . . . 13
- 2.2 Model Checking . . . 14
  - 2.2.1 Transition Systems & Deductive Logic . . . 14
  - 2.2.2 Our Implementation . . . 16

3 Modelling Systems **17**
- 3.1 Finite Automata . . . 17
  - 3.1.1 States & Propositional Atoms . . . 18
  - 3.1.2 Transitions . . . 19
  - 3.1.3 Machine Input . . . 19
- 3.2 Specifications . . . 19
  - 3.2.1 Kripke Structures & Temporal Logic . . . 20
  - 3.2.2 Linear-Time Temporal Logic . . . 21
  - 3.2.3 CTL . . . 22
  - 3.2.4 Fairness . . . 23
- 3.3 Policy Rule Logic . . . 23
  - 3.3.1 Propositional Datalog . . . 24
  - 3.3.2 Inheritance . . . 24

4 Symbolic Model Checking **26**
4.1 NuSMV Models . . . 26
4.1.1 Definition . . . 26
4.1.2 Data . . . 27
4.1.3 Transition Definition Statements . . . 30
4.2 Relationship with Finite Automata . . . 31
4.3 Relationship with Kripke Structures . . . 32

5 Design & Implementation **33**
5.1 Policy Format . . . 33
5.1.1 Inheritance Rules . . . 33
5.1.2 Privilege Rules . . . 34
5.2 Model Format . . . 34
5.2.1 Roles/Actions as Input . . . 34
5.2.2 Policy Privilege Format . . . 35
5.3 Joining Model & Policy . . . 35
5.3.1 DEFINE for Query Logic . . . 36
5.3.2 TRANS Filtering . . . 38

6 Case Study **40**
6.1 Example Design . . . 40
6.2 Analysis . . . 41
6.2.1 Abstracted Model . . . 41
6.2.2 Policy . . . 45
6.2.3 Bug Resolution . . . 47

7 Protection in Dynamic Systems **48**
7.1 Modeling Protection Systems . . . 48
7.2 Transfer of Rights . . . 50
7.3 Results . . . 57

8 Conclusions & Future Work **59**

Chapter 1

Access Control

Computer security includes topics such as authentication, confidentiality, and availability. An important component of a secure computer system is access control. An access-control policy is a description which outlines the actions that users can perform on resources within the system.

It is necessary to restrict privileges for different users to prevent misuse of the system, whether intentional or accidental. It is also necessary that a security system should not prevent users from gaining access as intended. Without proper limits on privileges, an entry-level programmer might accidentally ruin the configuration of a code version-tracking system, an employee could overwrite a file containing a schedule of shifts which was only meant to be read, or human resources employees could propose and approve their own raises electronically. *Access control* is the part of computer security which prevents the occurrence of these problems by restricting access to parts of the system based on the assignment of *privileges* or capabilities [9].

Once a user has been identified using authentication, the user's privileges can be determined using access control. The access control mechanism is separate from all other parts of security, and should only refer to external data in order to make decisions about privileges. Abstractly, the policy is a function that returns true or false given a request and the state of the system. Since the state of the system is ever-changing and policy rules can become convoluted, it is useful to be able to verify the privileges a policy grants and denies as time progresses.

As sensitive information systems – such as a medical record database – become more pervasive in society, the need for correct policies to govern access can only grow. The motivation for perfecting access-control policies range from privacy concerns, business protection, consumer protection, to even national security. We will present a system for using symbolic model checking to verify access-control policies paired with changing systems.

1.1 The Policy

1.1.1 Privilege Rules

An access-control policy is a formal description of the users, resources, and privileges. The policies can be directly represented as a matrix of subjects and resources, wherein each cell contains the actions a subject can perform on a resource. This representation of a policy can become exceedingly difficult to maintain manually, due to the fact that a realistic access-control policy may have to specify the privileges of hundreds of users over thousands of resources. Therefore, more practical representations are used in practice. It is common to simplify how the policy is described using general rules that describe how privileges are assigned to users. Regardless of the implementation or technique used, an access-control policy always relates to the matrix of privileges for the moment in time when the policy is queried.

There are two categories of techniques to specify the policy: discretionary access control (DAC) and mandatory access control (MAC). However, another technique called role-based access control (RBAC) fits neither of these models has been formulated [13]. Role-based access control is based on roles to which users are assigned to and the privileges assigned to roles. It is well suited for policies which must follow security principles such as separation of duty and least privilege [12].

1.1.2 Separation

The system which the access control regulates is otherwise governed by its own logic. The policy's job is to decide how to restrict actions on the system. The means or mechanism by

which access control is implemented is not important, as long as the mechanism can refer to a separate policy (or set of policies) to decide if an action by a user is permitted.

The policy is separate from the system in order to properly maintain access-control logic. This is due to the fact that scattering access-control logic throughout a system based on where it is needed makes it nearly impossible to maintain. Changes or bug fixes to access rules would require modifying scattered bits of the policy is difficult. It is also hard to see how parts of the policy fits together as a whole if they are not separate from system logic.

Least Privilege

A policy should follow good security principles. *Least privilege* is the concept that users should be assigned only enough privileges do their jobs. This implies that individuals with limited responsibilities should have limited privileges, due to the fact that they have less requirements. Users with greater needs should have more privileges. However, this principle goes not guarantee that individuals cannot use their privileges to harm the system.

Separation of Duty

Some critical or sensitive operations should never be performed by agents acting alone. One such act is the launching a missile from a nuclear submarine, which in the movies is always depicted as involving two distinct officers turning their respective arm-activation keys). Separation of duties is achieved by designing a policy so that certain tasks can only be performed by mutually exlusive roles. A classic example of separation of duties is separating the task of issuing a check between two roles: accounting clerk and account manager [12]. Without the proper separation of duties, it is possible for an individual to abuse a system by firing a missile at a whim or approve one's own expenditure.

1.2 Verification

Since access-control policies in the real world are not organized as unchanging, giant matrices, it is useful to be able to have assurances that the policy will behave as planed. To *verify* a policy is to make sure that it grants access to a system as expected. The expectations of a

policy are written as a *specification*, which is a set of statements that define the requirements of the policy. If the specification is correct and the policy satisfies the specification, then the policy is correct.

Due to the importance of assuring that a system is safe and upholding principles of security, it is necessary to verify the properties of a policy to assure that it grants privileges as expected. This is necessary since privileges are indirectly assigned to users through rules. Access-control policies are defined using rules which can have subtle and surprising behavior under the right conditions. Also, the direct verification of the policy is complicated by the fact that the privileges which are granted can change with time, since policy rules can depend on changing facts in the system itself. For example, a policy may grant a privilege only during a specific time range, or if a certain condition about the state of the system holds.

In access policy administration, it may be required to make changes to the policy itself. This happens if the roles or resources in the system change. However, it may be tempting to assume small changes in a policy will not yield unpredictable or dangerous results. It is thus necessary to check policies after changes are made. This requires automatic and rigorous verification, since an administrator cannot be expected to check a policy "by hand" whenever a change needs to be made, since several versions might be needed to find the right fit.

An access-control policy requires great care to ensure that it fits its specification, that it grants and denies the privileges without conflict, and that individuals can access the system resources as intended by the access control administrator. The requirements themselves are difficult to check for a changing system, since statements and relationships change in the system as time progresses. A different approach which proves more fruitful is model checking, which uses an abstraction of the system, includes the policy's privilege information as time changes, and a specification to verify the correctness of the system [5].

1.3 Classic Access Control Paradigms

1.3.1 Discretionary Access Control

Discretionary access control dictates the privileges users have on resources based on the identity of users and objects. Each users has authorizations which state which how each object is accessible through actions [13]. The traditional filesystem for an operating system uses DAC to specify which users and groups can perform actions on files based on the files' access attributes. For example, a file's owner can have complete access, while those users in a specific group may only read the file. The identities of approved users and groups are attached to each file, so that they can be accessed by the access-control mechanism of the operating system [13].

Due to the fact that the permissions for *each* resource can be set explicitly and independently, this makes it very difficult to maintain general access-control rules for a system that uses a DAC policy to decide permissions. In order to make a general change to the policy, the permission attributes of each resource must be maintained. In effect, the access policy is spread out over all resources (or an entire filesystem). The size of the policy is a challenge for rigorous policy maintenance [13].

1.3.2 Mandatory Access Control

Mandatory access-control policies do not assign privileges based on identity; permissions are instead based on the assignment of users and files to security levels. Classification levels are ordered so that one access level has greater restriction that another. Levels are assigned based on the trustworthiness of users or the sensitivity of information. For example, a user may only read resources which have a security level which is less than or equal to that of the user; on the other hand, a user may only write to resources which have a security level above or equal to that of the user. Informally, this can be described as an "upward" flow of information such that users with the highest level can read anything and users with the lowest level can only read resources with an equal level. The opposite is true for writing, although some MAC systems do not allow upward writing because users who write to a document with a higher level cannot read what they overwrite [13].

The classic example of MAC is that used by the government information classification hierarchy, which has the security levels (in decreasing over of sensitivity) Top Secret, Secret, Confidential, and Unclassified. Users with Top Secret clearance can read anything and only Top Secret users can read Top Secret documents. Resources with a lower security level are less restricted, while users with a lower security level are more restricted [13].

Information integrity is another system to which MAC can be applied. For example, the integrity of information is maintained by keeping information in unreliable documents to move to more reliable documents [13]. This can be described by keeping users from writing to documents with a higher security level and reading from documents with a lower security level. This keeps a user from reading information from a more unreliable document, and writing that information to a critical and reliable document. Thus, information flows downward to documents with lower integrity.

DAC policies cannot specify the restrictions on the flow of information like MAC policies can, which allows DAC and MAC to be combined in a system's access-control mechanism. However, MAC does not complement the weaknesses of DAC. MAC only specifies the direction of information; it is *too general*. One promoted alternative to discretionary and mandatory policies is *role-based access control*, which is used to specify rules for how users with roles can access resources.

1.4 Role-Based Access Control

Role-based access control (or *RBAC*) policy assigns privileges not to individual users but to roles, such as "engineer," "accountant," or "network administrator." These roles are then assigned to users, giving those users the privileges of their assigned roles. Classification is also given to resources in a similar manner so that the creation of a resource does not require the creation of new policy rules to manipulate the resource. Thus, making changes to the system do not require changes in the policy as well. This role-based approach also reduces the size and complexity of a policy while still allowing detail and power [13].

Although RBAC often simplifies the description of the access-control policy, the rules can still quickly become difficult to maintain. It is possible to simply compute the access-

control matrix from an RBAC policy and check for problems when constructing it. One example of a problem would be a contradiction in the system arising from both permitting and denying a user a specific privilege. Another problem might stem from allowing a user to circumvent an intended separation of duties by possessing multiple roles. If the system were static, verifying a policy for the system would amount to computing the single access table and verifying it using a logical specification. However, an actual system would have an ever-changing environment made up of users, resources, and role assignments. Taking the consequences of a dynamic system into account complicates the matter, especially when analyzing temporal behavior of the policy's rules [12].

1.4.1 Entities

The basic entities of an RBAC policy are *users*, *roles*, and *permissions*. These are abstractly assigned to set U for users, R for rules, and P for permissions. Users are assigned to one or more roles according to the *User Assignment* relation $UA \subseteq U \times R$. The permissions reflect the possible actions in the system.

This marks the difference between role-based and discretionary access control, which assigns permissions to users directly rather than indirectly, through roles [12]. The simplicity of assigning users to a few roles rather than individual privileges simplifies the process of administering access to a system.

Roles

Although sets of users can be assigned to roles, this should not be confused with groups in the DAC paradigm. Roles are both collections of users *and* privileges [12]. Roles are assigned to individuals to specify their responsilibities in a system.

Privilege Inheritance

An important part of role-based access control is the concept of *privilege inheritance*. A role may inherit privileges of other roles according to a partial ordering. Inheritance can be used for delegation and separation of duties. For example, a manager role can inherit privileges from a engineer role to show that the manager has more responsibility [12].

If a role inherits privileges from other roles, its set of privileges is a superset of the other roles' privileges. Since roles are sets of privileges, inheritance is defined as a partial ordering relation $\subseteq$: R^2 that defines inheritance.

Although inheritance adds more organization to an RBAC policy, it also makes verification more difficult due to the fact that privileges may "creep up" the role hierarchy in unexpected ways. It may also be tempting to naively design a hierarchy with as a tree with a unique root, giving a manager at the top of the hierarchy all the privileges which the less powerful roles possess.

Actions, Resources, & Privileges

Actions are not simply performed by users on resources, but are instead performed by users assigned to roles onto resources assigned to groups. For example, resources of the type "homework" can have actions performed on them, such as "submit", "grade", etc. A privilege is a paired action and resource which is assigned to a role, so that members of that role can performed the action on the resource.

Due to the fact that resources are tightly paired with actions, privileges can be considered as units, such as "assign homework" or "grade homework." Such a design follows the principle of data abstraction.

Queries

An access control request is a test that a user is given a privilege (and can thus perform an action on a resource). This is abstractly checking a permitted action for a user and a resource in the access control matrix. However, in RBAC, it is performed by an implementation mechanism which checks a policy composed of rules. The rules decide if a privilege is assigned or not, based on assigned roles and current circumstances and constraints. One language to write such rules is XACML, which is suitable for RBAC policy descriptions [6].

1.4.2 Concepts & Principles

The interaction between different entities in role-based access-control policies makes it suitable to support important security principles such as separation of duty and least privilege.

Verifying that these principles are upheld is necessary to administrating a policy which is designed to follow them. This can be difficult due to the nature of how role assignments and changes in the system due to time affect how privileges are granted.

Roles and inheritance play an important part in RBAC's handling of least privilege, Since users only receive privileges from their roles, an RBAC policy can be organized to have well-defined roles which have requires privileges. Inheritance makes the organization more clear, although inheritance can have subtle consequences that may not be caught by an administrator without proper verification [12].

Separation of duties is also well handled by RBAC since separate roles can be defined to handle different parts of a larger, critical task. However, problems do arise when a user can be assigned to two different roles, which in different contexts do not violate requirements for a policy. Such a case occurs in an academic institution where the separate roles of thesis advisor and thesis examiner can be assigned to professors. Although professors are supposed to be able to invoke both roles, it is important that no professor both advises and formally examines the same thesis. Adding separate contraints to possible configurations of role invocation would be necessary to avoid such a conflict [12].

1.4.3 Datalog

One method of expression RBAC policies is with a deductive logic such as Datalog [1]. A Datalog policy is made up of Horn clauses of the form $P_0(X_0) \leftarrow P_1(X_1), P_1(X_1), \ldots, P_n(X_n)$ where:

- P_0, ..., P_n are *predicates*.
- X_0, ..., X_n are tuples of terms, each of which can either be constants or variables.
- The left-hand side of the rule arrow is the *head* of the rule.
- The right-hand side is the *tail* of the rule.

Predicates correspond to tables of facts, where each row in the table is a tuple of constants which hold true for that predicates. $P(X)$ is considered a true fact if $X = \langle x_0, x_1, ..., x_n \rangle$ is a row in predicate P's table of facts. Datalog rules define how new facts about the tables of

a predicate are deduced given the tables of other predicates. Initially, there are predicates whose sets of facts known. Facts for other predicates are found through pattern matching with assignments. If there exists an assignment of constants to the variables in the rule such that each $P_i(X_i)$ holds true in the tail, then $P_0(X_0)$ also holds and the constant X_0 is a member of predicate P_0 table of facts. These valid assignments are computed recursively, until all possible assignments are added.

Here is an example policy rule written in Datalog:

$$Permit(R, \text{submit-paper}, P) \leftarrow Author(R), Paper(P), Phase(\text{submission})$$

This rule states that for each author R and paper P, R is permitted to perform action "submit-paper" on P during the "submission" phase. Permissions are deduced by matching variables R and P to constants in the tables of predicates *Author* and *Paper* and using those assignments to generate new facts for predicate *Permit*.

The input tables of a Datalog program is called the *knowledge base*. The predicates of those tables are called the intensional predicates and predicates whose tables are deduced from rules are called extensional predicates. For a policy defined by Datalog, the intensional predicates include *Permit*, and the extensional predicates describe the environment state.

Chapter 2

Related Work

Due to the fact that access-control policies are necessary to guard important data and systems, it is important that policies can be verified formally. Changing an access-control policy may be necessary for a changing system or a system with different requirements. However, new changes to a policy's rules can easily have unforeseen consequences to the results of queries to that policy.

2.1 Change-Impact Analysis: Margrave

Since the cost of poor policy administration is can be very high, policy verification is a valuable ability. It does not suffice to informally verify a policy by hand, because the rules of a large policy interplay subtly. Also, it is tempting to assume that making small changes to a policy will have limited consequences. It is possible to check a policy as a whole by hand after every change, but such an approach is time-consuming.

Instead, it is necessary to verify policies by algorithmic means. Margrave [6] is an access-control policy verifier which can test encoded properties of a policy and compute changes between two different policies which govern the same system. It verifies policies alone, without a model of the system, described in XACML using multi-terminal binary decision diagrams (MTBDDs).

Margrave works with role-based access policies written in XACML, a standard declarative language which describes access policies. A program which follows an XACML policy

matches sets of attributes and values to decide the result of access queries. For example, a request can have an attribute called "role" set to the value "student," and attribute "action" set to "read." A permit/deny answer can be specified by writing that requests of the form role=student, action=read, resource=grades will return *Permit*.

Margrave encodes these policies using MTBDDs, which are decision diagrams with several different *terminals*, which correspond to answers such as *Permit*, *Deny*, and *Not Applicable*. MTBDDs are run with sets of attribute-value pairs which represent queries. Running an MTBDD involves progressing from a start node in the decision diagram, moving from one node or another based on the presence of an attribute-value pair in a query. The result of an access query is the decided terminal. PMTBDDs are policies encoded in this manner. PMTBDDs, including individual rules, can be combined into bigger policies. Changes between policies are analyzed by modifying these structures and their results.

2.2 Model Checking

The access control research by Dougherty, Fisler, and Krishnamurthi [5] introduces formal analyses for role-based access-control policies and dynamic environments that use them. It is the theoretical basis for our project which is an implementation of the presented ideas. It demonstrates how a set of deductive logic rules and a transition system can be used to model a policy and an environment it regulates. The two can be merged into a single model, which can then be analyzed to verify the policy's properties.

2.2.1 Transition Systems & Deductive Logic

An *environment* is the set of facts or variables present in a system. A static environment does not change, and is an unchanging model of a system or set of facts. Real-world systems have information which changes as time progresses; a dynamic environment is the schema which describes the information as it changes. At a point in time, the dynamic environment provides a database of facts. This changing environment is modeled as a transition system where states are defined by the facts at different points in time.

A policy can be formally expressed with deductive logic. Datalog, a subset of Prolog, is

the language of choice for the policy due to the fact that it has nice, decidable properties. A Datalog program formally takes as its input an *extensional database* or *EDB*, of facts. From this initial set of information, new information is deduced using Datalog rules written as Horn clauses. The deduced information is called the *intensional database* or *IDB*; it is the output of a Datalog program. When used to write policies, the Datalog policy rules deduces the 3-tuples for the *Permit* predicate which decides privileges based on role, action, and resource. If a 3-tuple query is present in the set of the *Permit* predicate, then that query's result is "Permit."

A policy written in Datalog takes a static environment as its extensional database. The output intensional database provides a set of permitted actions. The new facts model answers to queries the access-control mechanism uses to permit or deny actions attempted by users. The original EDB facts which make up the environment and the IDB facts deduced by the Datalog policy make up the *access environment.*

However, this mathematical model is complicated by the fact that the environment is dynamic and changing. This means that the access environment is also dynamic since the deduced IDB facts change as the EDB facts change. The transition system which defines the dynamic environment is modeled as a transition system with EDB facts at each state; therefore, a new transition system can be modeled which unifies the transition system's states' EDB facts with the deduced IDB system. This new model shows both how the system and the user privileges change.

Simply merging the EDB and IDB facts do not correctly model a system with access control, since the privileges do not affect how the system behaves. Some transitions model user actions which change the state of the system. User actions are what are restricted by the access-control policy. Therefore, a correct model is produced by removing transitions which model user actions which are not granted by the deduced privileges at each state. So, if a state has a transition modeling a user action which is not shown to be permitted by the IDB facts merged with that state, the transition should be deleted. This new model is called the *dynamic access model*, which can be tested and analysed.

Other Analysis Methods

One method by Masood et al [10] RBAC testing methods are used to generate tests for implementations of RBAC policies. Their method uses a finite state model, but the model expresses the required behavior of the policy, not the environment. The model cannot be directly verified, since it is impactically huge without using heuristics to reduce the state space. However, the tests generated do not completely verify the policy, since they are simplifications of the actual model.

2.2.2 Our Implementation

Formally, our technique uses a very similar approach to verify access-control policies. We use transition systems to model the environment, although we specifically use finite state machines to model system behavior and Kripke structures to model the changing environment. Kripke structures use propositional facts instead of databases of tuples to keep track of changing information in the environment. This means that Datalog predicates are not used. Instead, propositional facts for privileges are deduced from propositional atoms which label states in the Kripke model.

Using temporal logic to verify Kripke models is well understood. It can be effectively used to express temporal properties of the sets of privileges granted by policies. Such models can be analyzed and simulated using symbolic model checkers to see if they satisfy a specification. The model checker we will use NuSMV, and will take advantage of the syntax of the input in order to merge a policy's rules and the original system model. By merging these two concepts, we will show that this static analysis technique can be but into practice..

Chapter 3

Modelling Systems

To define the behavior of a system which takes user input, we have used finite automata to abstract model behavior. To define the environment of the system and verify it, we must define a *Kripke structure*. However, this structure is produced in conjunction with an automaton which performs transitions based on an infinite stream of input. Since we need to model both user input and a changing environment, we will present both finite automata and Kripke structures.

We also deal with two kinds of logics: logics to specify requirements of the model, and logics to express rules of the policy. Temporal logic is used for specifications, while deductive logic is used to state policy rules.

3.1 Finite Automata

Definition. A **finite automaton** is defined by the tuple $\langle \Sigma, Q, Q_0, \delta \rangle$ where:

- Σ is the input alphabet.
- Q is the non-empty set of states.
- Q_0 is the set of start states, where $Q_0 \subseteq Q$.
- δ is the transition function which defines transitions from one state to another based on the input. $\delta :\subseteq Q \times \Sigma \rightarrow 2^Q$.

The automata we deal with are nondeterministic, meaning that the transition function returns a set of next states. This set can be the null set. Our definition of a automaton does not include a set of accepting states due to the fact that the automata we deal with cannot "accept" an input in the same way as a DFA or NFA. The model abstracts a system which runs forever, although since it is non-deterministic it is possible that the system "hangs" if it the set of "current" states becomes empty during a run. Although the automaton is non-deterministic, only a single state is kept track of when using model checker simulation and random behavior is assumed when more than one possible state is returned by δ.

The formal mathematical definition is important to the design of our project. However, an equivalent definition better suits our project in order to relate the theory behind our design and the implementation.

3.1.1 States & Propositional Atoms

For temporal logic, a *state* represents a the condition of a system during a moment in time. Each state has a finite set of *atomic propositions* associated with it which specify the facts which are true at that state in time. For example, one state might have atomic proposition p which represents the fact that it is currently daytime and q which is is present when a computer is idle. From state to state, facts can change just as they do from time to time.

The set of states is defined as the power set of the set of propositional atoms in the model. States in such a model are unique in the sense that no two states have the same set of propositions. This means that only one state presents a point in time where certain atomic facts are true or not true. The set of propositions of a state identify that state uniquely. They are refered as the *state variables* in model checkers such as NuSMV, which can handle more values than truth and false.

One practical consequence of this is that the number of states can explode when a large number of atoms are used. For example, a system with 10 atoms can have up to 1024 unique states; a system with one more atom can have twice as many. The use of binary decision diagrams has made the manipulation of such large state machines more practical enough to be used with hardware design verification. Despite the fact that model checking is limited by the state space, this limitation does not stand in the way of our implementation.

Finite automata are not labelled with atomic propositions; these are a part of the Kripke structure. When a finite automaton is simulated by hand with a model checker, propositions are simply used to define states, although the behavior of the automaton does not depend on these propositions. However, atomic propositions are used when building the automaton and the Kripke structure. Changes in state variables define both the automaton and the Kripke structure, although they are only used in Kripke structures for verification.

3.1.2 Transitions

A finite automaton takes an input which decides the possible transitions from a current state to next states. The future states in simulation depend on the sequence of input of the state machine . The input symbol of the state machine labels transitions in the state machine to represent the input and output of the transition function. During simulation with a model checker, the input variable behaves non-deterministically to simulate actual input.

3.1.3 Machine Input

Each transition is labelled with only a single input symbol. Although the δ only takes a single member of the input alphabet Σ, it is useful to define Σ as a Cartesian product of several alphabets to represent several simultaneous inputs. For example, the definition $\Sigma = \Sigma_0 \times \Sigma_1 \times \ldots \times \Sigma_n$ can be used to signify that the FSM takes several different inputs, where each Σ_i is an alphabet for a specific input. This definition is used to model a state machine with several distinct inputs. This definition is still equivalent to the original one and does not increase the power of the automaton; it is simply is easier to work with.

3.2 Specifications

Temporal models described as finite automata are written to satisfy properties written in temporal logic. In logics without temporal operators, facts about a model are a set which cannot change. However, facts in temporal models can change. Temporal logics can describe such changes using temporal connectives which describe the relationships between facts as time progresses or the possible existence of facts in the timeline.

Two such logics include linear-time temporal logic (LTL) and computation tree logic (CTL). LTL and CTL treat branching in the Kripke structure's computation tree differently. Both logics have different temporal operators and neither is a subset of the other. However, both can be used to verify the same kind of model. This allows flexible specifications to be written in practice since one logic can be used to verify properties which the other cannot express [4].

3.2.1 Kripke Structures & Temporal Logic

CTL and LTL express properties of states based on the labeled atomic propositions of each state in a transition system. This system is called a *Kripke structure*[4], which is used to model concurrent systems with finitely many states.

Definition. A **Kripke structure** M over AP is a tuple $M = \langle S, S_0, R, L \rangle$ where

- S is a finite set of states.
- $S_0 \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is the total translation relation.
- $L : S \rightarrow 2^{AP}$ is a fucntion that labels states with a set of atomic propositions true in that state.

The Kripke model is defined by model checkers by NuSMV by a plain-text input which defines variables (which correspond to atomic propositions), restrictions on transitions, and other constraints.

A Kripke structure is a model which defines how state changes in a concurrent system. Each state has a set of atomic propositions which express facts about the current state of the system. Transitions in the Kripke model signify changes in the system over time although transitions which start and end at the same state are allowed. The transition relation is total, meaning that every state has a transition to another state or itself.

A model has a set of system variables V, each of which has a finite set of possible. Atomic propositions are singular facts about the system state. They are expressed as $v = d$ where

$v \in V$ and $d \in D$, where D is the finite set of possible values for v [4]. For example, a boolean system variable x is defined by two mutually exclusive atomic propositions: $x = True$ and $x = False$. At each state, a *state formula* is a propositional expression which is composed of atomic propositions and propositional logic operators, such as $\wedge, \vee, and \rightarrow$. State formulas express properties about individual states in a static environment, so that each state works as a model in propositional logic. These formulas are used by more expressive temporal logic formulas which express properties of a Kripke structure as a whole.

3.2.2 Linear-Time Temporal Logic

Running a Kripke structure is performed by "walking" a model M, moving from state to state. This simulates a run of the abstracted system. *Linear-Time Logic*, or LTL, formulas to express the changes in the atomic propositions along an infinite path through the Kripke structure. An infinite path $\pi = s_0, s_1, s_2, \ldots$ exists such that for every $i \geq, (s_i, s_{i+1}) \in R$. This means a path must be made up of transitions from a start state to future states. The subpath of state π starting from state s_i is denoted by π^i [4].

Changes in the sets of atomic propositions which label states are expressed using temporal operators **X** ("next"), **F** ("future"), **G** ("globally"), **U** ("until"), and **R** ("release"). If f is an LTL formula and Kripke structure M satisfies a formula f from its start state, this is denoted by the statement $M \models f$. For a path π, we can write $M, \pi \models f$ to express that the path satisfies the formula. For a model M, a path π, propositional logic formul p, and LTL formulas f_1 and f_2:

$M, \pi \models p \Leftrightarrow p \models L(s_0)$.

$M, \pi \models \neg f1 \Leftrightarrow M, \pi \not\models f_1$.

$M, \pi \models \mathbf{X} f_1 \Leftrightarrow M, \pi^1 \models f_1$.

$M, \pi \models \mathbf{F} f_1 \Leftrightarrow$ there exists a $i \geq 0$ such that $M, \pi^i \models f_1$.

$M, \pi \models \mathbf{G} f_1 \Leftrightarrow$ for all $i \geq 0$, $M, \pi^i \models f_1$.

$M, \pi \models f_1 \mathbf{U} f_2 \Leftrightarrow$ there exists a $k \geq 0$ such that $M, \pi^i \models f_2$ and for all $0 \leq j < i$, $M, \pi^j \models f_1$.

$$M, \pi \models f_1 \mathbf{R} f_2 \Leftrightarrow \text{for all } j \geq 0, \text{ if for every } i < j M, \pi^i \not\models f_1 \text{ then } M, \pi^j \models f_2$$

$\mathbf{X}\ f$ means that f holds in the path starting from the next state in the current state, $\mathbf{F}$ f means that f holds in a subpath, and $\mathbf{G}\ f$ means that f holds for all subpaths of a path. Binary operator $f_0\ \mathbf{U}\ f_1$ means that formula f_0 holds for all subpaths at the very least until f_1 holds. $f_0\ \mathbf{R}\ f_1$ means f_1 holds either for all subpaths or until the subpath before f_0 holds.

Here are a few examples of LTL formulas:

- $\mathbf{G}(p \leftarrow \mathbf{F}(p \wedge \mathbf{X}q))$ expresses the property that for all paths, if the current state is labelled p, then a future state down the path will be labelled p, immediately followed by a state labelled by q.

- $\mathbf{GF}(busy\mathbf{U}(done \vee cancelled))$ states that *busy* will hold until either *done* or *cancelled* holds. This property must hold infinitely many times, which is states by $\mathbf{GF}f$, where f is the property.

- $\mathbf{F}(p\mathbf{R}q)$ means that sometime in the future, q will hold until the state before p holds or it will hold forever.

LTL can be used to express a great deal of specifications for model checking. However, it cannot be used to express that something *may* happen in the future, since a path is a determined simulation of a model.

3.2.3 CTL

Computation Tree Logic treats time differently than LTL. CTL describes properties of computation trees rather than just paths. In such trees, nodes are states, and the subnodes of a state are the finite set of next states which can follow the current state during simulation as defined by the transition relation. A Kripke structure is unwound into a tree of infinite height. There may be many infinite paths down a tree at a current state. Therefore, it is possible to quantify paths, so that properties may hold for at least one or all subpaths from a node.

CTL formulas are similar to LTL formulas, except that temporal operators must be quantified with either **E** or **A**. Here are some examples:

- **AX**p states that p must hold for all states after the current state. This is the same as stating that **X**p must hold for all paths from the current state.

- **EG**q states that q must hold for every state in a path from the current path. This is the same as stating that there exists a path from the current state where **G**q holds.

- **EFA**$[p\mathbf{R}q]$ states that there exists a path from the current state with a future state which has all possible subpaths satisfying the property $p\mathbf{R}q$.

3.2.4 Fairness

When performing model checking, it is sometimes necessary to be able to restrict the kinds of paths that are considered when testing LTL and CTL properties. For example, testing a mutex lock could require constraints that guarantee that certain conditions occur, such as guaranteeing that when a program enters a mutex, it must eventually release it at an unknown time in the future. Due to the fact that time in Kripke structures is infinite, this requirement must be expressed using *fairness constraints*.

A fairness constraint is a logic formula that expresses a condition which holds for a subset of the states in the model. These states are fair states which are labelled with sets of propositions which satisfy the constraint. *Fair paths* are infinite paths along the Kripke structure which cross the fair states infinitely often. When used in model checking, only fair paths which satisfy this requirement are considered in LTL and CTL verification. This way, certain properties can be guaranteed to happen infinitely often in specifications in order to accurately model systems [8].

3.3 Policy Rule Logic

The policy contains rules which state how privileges are granted given the environment. The type of logic used defines how the policy is stated and how privileges are deduces. Previous work[5] has shown that Datalog is suitable for describing policies. However, we have used propositional deductive logic to describe policies, since it fits well with Kripke structures and model checking.

3.3.1 Propositional Datalog

Since our goal is to check Kripke models which have atomic propositions, policy rules are used to make a new Kripke model with added atomic propositions which correspond to all privileges allowed given the current environment. Datalog is not immediately suitable since Datalog deduces tables and Kripke model states are labelled with atomic propositions. Due to the fact that the set atomic propositions of a Kripke model is finite, we can use a less complicated logic to express policies which deduce the same amount of information about privileges.

Propositional Datalog is a form of Datalog which deduces atomic propositions rather than rows in tables. Propositional rules take the form $p_0 \leftarrow p_1, ..., p_n$, where each p_i is an atom. If the logical conjunction of the truthness of the propositions in the tail hold, then the head proposition is true as well. This is a suitable method to extend Kripke models with privilege information, since the set of propositions which label a state in the Kripke structure can be used as the knowledge base for the propositional rules. Deduced atomic propositions for each state's existing set of propositions can be used to extend the Kripke model with more information.

In our case, we do not have to strictly use Horn clauses. Instead, we can deal with rules of the form $p \leftarrow q$, where q is a boolean expression. Model checking systems such as NuSMV can deal with expressive formulas; therefore, there is no need to restrict the format of such rules when defining policy logic.

3.3.2 Inheritance

Role inheritance defines a partial order such that if one role r inherits the privileges of role r', then the set of privileges granted to r must be a superset of those granted to r'. In Datalog, this relation would be defined by the clause $Permit(r, A, R) \leftarrow Permit(r', A, R)$. It is unnecessary to check at runtime the inheritance relation since it is already encoded in the policy by the rule itself.

Similar rules can be written in propositional rules which have the same effect. For every action a, the rule $Permit_{r,a} \leftarrow Permit_{r',a}$ describes the inheritance of the privilege of r to

perform a.

Chapter 4

Symbolic Model Checking

Our goal is not to implement model verification from scratch, but to instead use verification by a practical means. NuSMV is an mature open source model checker which uses Kripke structures and temporal logic to perform verification of abstracted concurrent systems. Since it is open source, it can be modified to fit our needs. However, NuSMV's syntax is flexible enough to suit our needs without modification.

4.1 NuSMV Models

NuSMV is an open source symbolic model checker suitable for testing a Kripke structurel with LTL or CTL. It can also be used to simulate a model by hand as though it were a finite automaton. It takes as its input a temporal model which is user-friendly and converts it to a formal finite automaton. This automaton's states are labeled with atomic propositions during construction; these labels are used to produce the Kripke structure. The actual implementation that NuSMV uses is not relevant to our project. However, the syntax and semantics that NuSMV uses to convert the user's input into a model is of great importance [3].

4.1.1 Definition

Definition. A **finite-state machine** (or FSM) handled by NuSMV is defined by the tuple $\langle A, Q, Q_0, I, R \rangle$ where:

- A is the set of state atomic propositions (derived from variables and their possible values). Each input variable A_i has a set of possible values D_{Ai}.
- Q is the set of states where $Q \subseteq D_A = D_{A0} \times D_{A1} \times ... \times D_{An}$.
- Q_0 is the subset of start states $Q_0 \subseteq Q$.
- I is the set of input variables which label transitions. Each input variable I_i has a set of possible values D_{Ii}. The set of possible inputs for the model is Cartesian product $D_I = D_{I0} \times D_{I1} \times ... \times D_{In}$.
- $R \subseteq Q \times D_I \times Q$ is the transition relation which defines transitions from state to state.

A NuSMV model is mainly composed of variables, restrictions on state transitions, fairness conditions, and the logical specifications written in LTL or CTL. The model also contains extra information about synchronous/asynchronous simulation and separate modules which represent systems running in parallel; however, these features are not relevant to our project.

The specifications are not a part of the model, but are used to verify that the model satisfies expected properties written in temporal logic. At the very least, a NuSMV model consists of a set of variables and nothing else. In such a model, the computed automaton consists of the set of all possible sets (the power set of the variables) and all possible transition, so that every state has a transition to every other state and itself. The modeled system is completely unrestricted due to the fact that it models a system where anything can happen.

The NuSMV model only becomes useful after restrictions are added to the model's transition. Each individual restriction to the system acts as a filter for the transition relation R. By specifying restrictions, unwanted transitions and unreachable states are removed from the model.

4.1.2 Data

NuSMV syntax relies heavily on logical expressions which rely on the facts of a state (variables) and the labels of transitions (input variables)

Types

Rather than simply using boolean true/false propositions, NuSMV is equiped to handle more interesting data types, such as enumerated constants, small integers, and arrays. Here are some of the types which can be assigned to variables:

Booleans Expressions type `boolean` have a value of either `0` or `1`. They are the types returned by logical expressions and are the true types manipualted by NuSMV.

Integers Expressions can have integer values within a finite range. Integers can be used in arithmetic expressions to compute new values (as can booleans).

Constants Enumerated constant symbols can be used in NuSMV, although they do not have numeric values associated with them as in C/C++. However, constants can be compared for equality and inequality, such that two constants are equal if and only if they are the same symbol. Therefore, all constants are distinct.

Sets Certain expressions in NuSMV can have a set type, which contains a number of other values. Sets do not have to contain values of the same type, so the set $\{0, 3, Up, Down\}$ is valid.

Expressions can have any value. Variables on the other hand must have explicitly defined types. However, variables must have an explicit type or a defined set of possible values. Although NuSMV handles types other than boolean propositions, NuSMV reduces all types and variables to booleans internally.

State Variables

State variables model the facts which are true or false in the system at a specific point in time. Their values decide the set of possible transitions from the current state based on the transition rules defined for the model. If a state variable has no transition restrictions, it randomly changes value from state to state.

```
MODULE main
   VAR
     a := {0,1,2};
```

```
ASSIGN
  init(a) = 0;
  next(a) = (a + 1) \% 2;
```

The above code defines a simple model with a single variable a which can have the values 0, 1, or 2 as defined by the `VAR` statement. When the machine is run, the initial state of a is 0, and after every transition the value is incremented by 1 unless it wraps around in value back to 0. The behavior of state variable a is fully defined by the `ASSIGN` statement, which is explained in more detail later.

Input Variables

Input variables are similar to state variables except for a few important differences. First, the value of an input variables is assigned during runs of the system randomly, so they cannot be assigned values like state variables. This makes them suitable for simulating random model input.

Second, input variables label transitions in the finite-state machine. This means that the values of input variables can be refered to when filtering transitions with the `TRANS` syntax (explained below).

Aliases

It is possible to define new identifiers in NuSMV which work as aliases for expressions. This is done using the `DEFINE` statement which creates an alias for an expression:

```
DEFINE symbol := expression;
```

The defined symbol inherits the type of the expression. Any occurance of the symbol can be replaced with the expression without any change to the finite-state machine (and vice-versa). Symbols defined this way do not increase the size of the finite-state machine since it does not introduce new variables into the model. They do not make NuSMV more powerful, but they do make NuSMV code more clear by replacing redundant expressions or giving expressions meaningful names.

4.1.3 Transition Definition Statements

Transitions are defined by statements such as `TRANS` and `ASSIGN`. These statements are used to restrict what transitions from each state should be allowed, although they do so in very different ways.

`ASSIGN` statements

Assignment-sytle syntax using `ASSIGN` statement is the most readable. `ASSIGN` is used to define the next value of a state variable given the current variables (which can be either state or input variables). The following is a restating of the above example with a slightly different definition.

```
01: MODULE main
02:   VAR
03:     a := {0,1,2};
04:   ASSIGN
05:     init(a) = 0;
06:     next(a) = case
07:         a = 0 : 1;
08:         a = 1 : 2;
09:         1     : 0;
10:     esac;
```

In this definition, variable a changes value according to the sequence 0,1,2,0,1,2,0... using the `ASSIGN` statement. Line 5 states that the first value of a is 0. Note that there could be more initial values by replacing 0 with a set such as {0,1}. Lines 6 through 10 define the next values of a given the current state. Lines 7 and 8 show that the next value of a should be one more than the current value if the value of a; otherwise, the bottom-out case is used on line 9 and the value of a is set to 0 again. If other state variables are added, they can also have their individual `ASSIGN` statements.

Unfortunately, `ASSIGN` statements are formatted to resemble procedural code or function definition by case. This makes it difficult to define more than one variable at a time for a single case. `ASSIGN` statements are also less powerful and flexible than `TRANS` statements which explicitly state restrictions on transitions in the whole model.

TRANS statements

The TRANS statement has the following syntax:

```
TRANS simple-expression;
```

According to the NuSMV 2.4 User Model[2], the `simple-expression` is a boolean expression used to test if existing transitions should be included in the model. Only transitions which satisfy the expression remain. The `simple-expression` is used to directly filter transitions from the truth of the expression given the transition labels and the propositions of the current and next states of the transition. Here is an example TRANS statement:

```
TRANS (a & b) | (next(c));
```

This example restricts all transitions in the finite-state machine's transition relation to those which either have both `a` and `b` true in the current state or `c` true in the transition's next state.

Note that TRANS statements work as filters for the entire set transitions for the model. This means that they can affect other statements such as ASSIGN. This power that TRANS statements make them very useful to restrict model behavior by simply adding TRANS statements rather than modify the existing code.

4.2 Relationship with Finite Automata

The NuSMV model defines both a finite automaton and a Kripke structure. Since the NuSMV received input but does not accept it, this automaton has no accepting states. However, the start states of the NuSMV model does correspond to the automaton's start states. NuSMV produces a non-deterministic automaton; during interactive simulation, it is possible to select one of many next states from the current state and the input which labels the transition.

The input is also an infinite string of symbols. NuSMV models the input alphabet from the set of input variables. The automaton's input alphabet corresponds to the Cartesian product of the possible values of each of the input variables in the model. Therefore, the

input model's transitions are labelled with tuples of symbols which correspond to input variable values.

4.3 Relationship with Kripke Structures

It is harder to derive the Kripke structure from a NuSMV model than it is to derive the finite automaton. This is because Kripke structure transitions are not labelled; only the states can be labelled. However, the Kripke structure still needs to refer to input variables in the NuSMV model to perform model checking, since the input variables can be refered to in logical specifications.

The NuSMV user manual states that input variables can be defined just like state variables, except for restrictions on where input variables can be refered to. Foremost, input variables cannot be used in `ASSIGN` statements for initialization or assignment, although they can be used as conditions for the assignment of state variables. Input variables can be treated as state variables whose values cannot be defined deterministically.

Due to the fact that NuSMV treats input variables as random state variables, they are included in the Kripke structure's definition in the way fashion. Input variables in the automaton label transitions, but in the Kripke structure they label the start state of transitions. However, in the automaton the transitions from a single state can be labelled with different input symbols. This requires that states in the NuSMV model and the automaton must be duplicated in the Kripke structure, where each duplicate is labelled with a distinct set of atomic propositions which correspond to the input variables.

In practice, the Kripke structure will not be that large since we intend to prune unwanted transitions from the NuSMV model. The encoding that NuSMV uses for models behind the scenes also allows us to work without worrying about the state space.

Chapter 5

Design & Implementation

Our tool for access-control policy verification takes as input a NuSMV model for the environment and a policy file. It outputs a merged dynamic access model which can be used for verification using NuSMV model checking. The description of the input files, the algorithms involved, and the results of our tool can be used to re-implement the tool without the presence of source code.

5.1 Policy Format

The policy will be made up of two different kinds of rules. There are inheritance rules, which define how roles inherit privileges from other roles, and there are privilege rules, which define the condition necessary for a role to be granted a privilege. The policy is a plain text document which is later merged with a NuSMV model of the dynamic environment of the policy. The policy refers to *role identifiers* and *action identifiers* which also show up in the model definition.

5.1.1 Inheritance Rules

An inheritance rule has the following syntax:

```
<role> inherits <role> [, <role>[, ...] ].
```

The left side of the rule has the identifier of the role which inherits privileges, while the right side contains identifiers of roles from which privileges are inherited, separated by commas.

An inheritance rule can only be written once for a single role.

Since role inheritance is a partial order, there can be no circular definitions for role inheritance. This means that a role cannot depend on itself for inherited privileges and that there must be at least one role which does not inherit any privileges.

5.1.2 Privilege Rules

Inheritance rule syntax takes the following form:

```
Permit(<role>, <action>) := "<boolean expression>".
```

The rule states that the given role is permitted to perform the given action when the boolean expression (which depends on the dynamic environment) evalutates to *true* or 1. At most, only one rule can be written for every distinct role/action combination. If a rule is not present for a particular role/action, that privilege is by default denied.

5.2 Model Format

The NuSMV model defines the dynamic environment of the policy and abstracts the concurrent system which is to be restricted by access control. In order to keep the model simple, separate modules and asynchronous transitions are not supported by our technique. The only module in the NuSMV model input file should be `MODULE main`.

5.2.1 Roles/Actions as Input

Two input variables must be defined inside the NuSMV model. This is necessary to simulate user actions which change the system. The declarations take the following form.

```
IVAR:
Role : { None, <role>[, <role>[, ...] ] } ;
Action : { None, <action> [, <action> [, ...] ] } ;
...
```

This defines that the input variable for the current role can either be `None` or another identifier which represents a role in the policy. The action is also similarly defined. The combination of the action and role determines when the system must react to user input.

Notice the `None` symbol. This is a special constant to our policy/model merging tool (but not to NuSMV). When the input includes `Role = None` and `Action = None`, the meaning is that there is currently no user action. This input is useful to simulate transitions which do not depend on user action. It is also necessary to have such an input in case that *no* user actions are allowed, which can certainly be the case. If no actions are allowed, then all transitions which are labelled with unwanted actions will be removed from the merged model. However, the `None` action is allowed since it is null. Since it does not make sense to have a role perform no action, the role can also be `None`. The merged model takes into account the fact that `Role` and `Action` should only be `None` at the same time. The merged model enforced this restriction.

5.2.2 Policy Privilege Format

If the model needs to test that a particular action is granted to a role somewhere in the model, the tool will insert a defined identifier which holds this value. The identifier takes the format `Permit_Role_Action`, where `Role` and `Action` are replaced with role and action identifiers in the policy and model. This identifier is not a variable, but a defined alias for the boolean expression of the corresponding policy privilege rule.

It is important to note that the privileges do not need to be refered to in the input NuSMV model when writing system logic, since the transitions with denied actions will already be removed from the model. The identifier can be refered to in the temporal logic specification to verify the state of granted privileges.

5.3 Joining Model & Policy

In order to merge the policy to the model, two things must happend. First, the policy's privilege logic must be added to the model. Next, transitions whose action labels contradict the policy must be removed.

The input model only has code in `MODULE main` and statements can be included in any order. Therefore, extra statements can be appended to the model to add more information or definitions without having to modify the existing contents.

5.3.1 DEFINE for Query Logic

As previously mentioned, NuSMV `DEFINE` statements create aliases for expressions without adding variables to the model. Therefore, it is natural to define policy rules as `DEFINE` statements.

Definition 1. If there is a policy rule `Permit(`R`, `A`) := "`x`".` , where R is a role identifier, A is an action identifier, and x is a boolean expression which states the condition for the rule, the following statement is appended to the input NuSMV model:

`DEFINE Permit_`R`_`A `:= (`x`);`

This states that alias `Permit_R_A` is true if and only if the condition x for the privilege holds. This adds labels to the model's Kripke structure which can be used in verification.

If the rule is not present for a role and action in the policy, the privilege information still needs to be included. In such a case when the expression x is replaced with the constant expression (0) which represents falsity, since privileges are by default denied. This guarantees that privileges are well-defined in the policy, and that

Inheritance

Inheritance needs to be a transitive, partial relation. Inheritance is expressed by extending the `DEFINE` statements.

Definition 2. If a role R_0 inherits from roles R_1 through R_n, then for every action A the following *privilege statement* will be included in the merged model:

`DEFINE Permit_`R_0`_`A `:= (`x`) | Permit_`R_1`_`A ` | ...| Permit_`R_n`_`A`;`

where x is the boolean expression given in the policy rule `Permit(`R_0`, `A`) := "`x`".` , or 0 if the rule is not present and defaults to falsity/denial.

This defines the alias for the privilege of role R_0 to perform action A, which holds true if and only if condition x holds or an inherited role holds the same privilege. This is clear since the permission for R_0 and action A is defined as the logical disjunction of the condition x and the permissions of the inherited roles with the same action.

The tool maintains the constraint that there can be no cyclical inheritance: a role cannot directly or indirectly inherit its own privileges. However, it is necessary to prove that inheritance defined in this manner is transitive.

Theorem. *Modelled role privilege inheritance is transitive. This is true according to the policy's semantics and the merged model's defined properties:*

(i) If role R_a inherits from R_b and R_b inherits from R_c, then role R_a inherits from R_c.

(ii) The merged NuSMV model has the property $\texttt{Permit_}R_c\texttt{_}A \rightarrow \texttt{Permit_}R_a\texttt{_}A$ for every action A in the policy.

Proof. Property [i] is present in the policy using role inheritance rules of the form:

R_a `inherits` R_b`,....` R_b `inherits` R_c`,....`

The tool adds this information in the merged model. Due to definitions 1 and 2, the following rules must be present in the merged model for every action A:

`DEFINE Permit_`R_a`_`A `:= (`x_a`) | Permit_`R_b`_`A ` | ...;`

`DEFINE Permit_`R_b`_`A `:= (`x_b`) | Permit_`R_c`_`A ` | ...;`

Due to the semantics of NuSMV expressions, this means that the following properties hold in the NuSMV model at each state (in the Kripke model):

$\texttt{Permit_}R_b\texttt{_}A \rightarrow \texttt{Permit_}R_a\texttt{_}A$

$\texttt{Permit_}R_c\texttt{_}A \rightarrow \texttt{Permit_}R_b\texttt{_}A$

Since logical implication is transitive, the following property must also hold:

$\texttt{Permit_}R_c\texttt{_}A \rightarrow \texttt{Permit_}R_a\texttt{_}A$

Since this property holds for every action A, modelled privileges are inherited transitively. □

5.3.2 TRANS Filtering

After the states are labelled with privilege atomic propositions, it is possible to remove restricted transitions based on this added information. We have determined that `TRANS` statements are appropriate for this task, since they are used to filter unwanted transitions from the NuSMV model's transition relation R.

Null Actions

Note that the Role and Action input variables can have the value `None`. When no action is performed, the transition represents a *null action*. To make the model correctly, we must remove action transitions where one input variable is `None` but the other is not. The following `TRANS` statement removes all transition where only one input variable is equal to `None`:

```
TRANS !(Role = None xor Action = None);
```

This statement eliminates transitions when one variable is equal to none but the other is not. This statement applies this restriction to all transitions in the model, which means that the original model code does not have to be interfered with to apply such a restriction. Instead, this statementsimply needs to be appended to the merged NuSMV model.

Denied Actions

Likewise, transitions based on actions which are not permitted are removed by `TRANS` statements. Granted policy privileges are represented by atomic propositions of the form `Permit_`R`_`A, which is true if the policy grants the right for a role R to perform action A. For a role R and action A, the policy's privilege results `Permit_`R`_`A is enforced by the following statement:

```
TRANS !((Role = R & Action = A) & !Permit_R_A));
```

This statement ensures that no transition can be labelled with input `Role =` R and `Action =` A if the role R is not granted the privilege to perform action A at the start state of the transition.

This kind of statement is appended to the merged model for every modelled role and action in the policy. This removes all denied actions from the model based on the results of the policy.

Chapter 6

Case Study

Having designed and implemented our tool for access-control verification, we tested the tool using case studies. We present an example case study which demonstrates merging the policy and model, NuSMV verification, and debugging the policy using NuSMV's results.

6.1 Example Design

Our case study abstracts an inventory/account system in an organization which must maintain a store of supplies. In the organization, employees request supplies for the inventory by submitting requests. An inventory manager reviews requests to order more supplies. A request is either rejected or accepted and written as an order. Since the inventory manager only keeps track of what supplies are needed and not the costs, the order is approved by an account manager. An account manager must approve orders before they are sent to be purchased by a supplier. An executive manager keeps track of all operations and can write inventory orders or approve orders if necessary.

The role organization is as follows:

- Employees are at the bottom of the hierarchy and do not inherit any privileges.
- Inventory managers inherit the ability to request orders, an ability inherited from the employee role.
- Account managers also inherit privileges from the employee role.

- Executive managers inherit privileges from inventory managers and account managers, reflecting their control of the organization.

In practice, an access-control administrator will write a policy for an existing model of the system. The policy can be merged with the tool to produce a model which can be verified using a temporal logic specification.

One property that an administrator would want to ensure is *separation of duties*, which is the property that certain important tasks require different roles to perform a single task. The inventory/account example can be verified to test this property.

6.2 Analysis

In the abstracted model and policy, these are the roles:

6.2.1 Abstracted Model

The system's model must include the list of

The roles in the system are: $Employee$, $InventoryMgr$, $AccountMgr$, and $ExecMgr$. The are declared in the model of the system, but show up in the policy's rules as well.

The actions in the system are as follows:

- MakeRequest: an employee requests a new supply.
- WriteOrder: an order is written from a request.
- RejectRequest: a reqest is rejected.
- ApproveOrder: an order is approved.
- RejectOrder: an order is rejected.
- SubmitOrder: an approved is sent, so the system is reset.

Variables

This section includes the declaration of the main module, state variables, and input variables. The state variables abstract information about the current request and order. In order to keep track of the role which was invoked by writing an order and approving them.

```
MODULE main
   VAR
        request-box    : {None, 1,2,3};
        order-type     : {None, 1,2,3};
        order-approval : boolean;
        order-done     : boolean;
        order-writer   : {None, ExecMgr, InventoryMgr, AccountMgr, Employee};
        order-approver : {None, ExecMgr, InventoryMgr, AccountMgr, Employee};
   IVAR
        Role    : { None,
                    ExecMgr,
                    InventoryMgr,
                    AccountMgr,
                    Employee};
        Action  : {None,
                    MakeRequest,
                    WriteOrder,
                    RejectRequest,
                    ApproveOrder,
                    RejectOrder,
                    SubmitOrder};
```

Assignment Statements

Next are the assignment statements for state variables. The model's initial state is a null state where no requests or orders have been written. The `init` statements define th intiial states of variables. The `next` states show how individual state variables change value from state to state. The first `next` statement defines the behavior of `order-writer`, which defines which role was used to write the order.

According to the code, if `order-done` is set to true the `order-writer` should be assigned to `None`. If the current user action is `WriteOrder`, the value is instead set to the symbol representing the current role (which is the value of the input variable `Role`). Otherwise, `order-writer` does not change value. After this rule, the statements for other state variables follow.

```
ASSIGN
    init(request-box)       := None;
    init(order-type)        := None;
    init(order-approval)    := 0;
    init(order-done)        := 0;
    init(order-writer)      := None;
    init(order-approver)    := None;

    next(order-writer) := case
            order-done : None;
            Action = WriteOrder : Role;
            1                   : order-writer;
        esac;

    next(order-approver) := case
            order-done : None;
            Action = ApproveOrder : Role;
            Action = RejectOrder  : None;
            1                     : order-approver;
        esac;

    next(request-box) := case
            order-done : None;
            Action = MakeRequest  : {1,2,3};
            (Action = RejectOrder | Action = RejectRequest)
                                  : {None};
            1                     : request-box;
        esac;

    next(order-type) := case
            order-done : None;
            Action = WriteOrder   : request-box;
            Action = RejectOrder  : None;
            1                     : order-type;
        esac;

    next(order-approval) := case
            order-done : 0;
            Action = ApproveOrder : 1;
            Action = RejectOrder  : 0;
            1                     : order-approval;
        esac;

    next(order-done) := case
            order-done : 0;
            Action = SubmitOrder  : 1;
            1                     : order-done;
        esac;
```

Specification

The last part of the model includes the specification information. The `FAIRNESS` constraint restricts the set of temporal paths to be verified to those where `order-done` is true infinitely often. This means that the system should not be stuck to infinity waiting for a user to submit a role.

For the purpose of this demonstration, the specification only has one requirement written as an LTL formula. It states that the role which was used to write an order cannot also be the role which has approved an order (`order-writer != order-approver`). This states a separation of duties requirement. The second subformula `order-writer = None` takes into account that `order-writer` is set to `None` as a null value; this part is necessary to take the null values of state values into account, since the system must be reset when the order is submitted and when the system is initialized.

```
FAIRNESS
    order-done;
LTLSPEC
    G((order-writer != order-approver) | (order-writer = None));
```

The parts of this model is written to the file `pay.smv`, which is a valid NuSMV file. When NuSMV is run with the file, the specification fails during verification and a counter-example is shown. Here is the start of the counter-example from NuSMV:

```
-- specification  G (order-writer != order-approver | order-writer = None)
      is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  request-box = None
  order-type = None
  order-approval = 0
  order-done = 0
  order-writer = None
  order-approver = None
-> Input: 1.2 <-
  Role = ExecMgr
  Action = ApproveOrder
-> State: 1.2 <-
  order-approval = 1
  order-approver = ExecMgr
...
```

The counter example lists changes in variable states and loops in the infinitely long execution paths. The initial state 1.1 shows that all state variables are either set to `0` or `None`. The input for transition 1.2 from state 1.1 to 1.2 is `Role = ExecMgr` and `Action = ApproveOrder`. Notice that this action does not make sense since the order cannot be approved before it is written from an approved request. This transition is possible due to the fact that no actions are eliminated, since the policy has not yet been merged to the model. Therefore, the specification is not satisfied.

6.2.2 Policy

The first policy we are checking is `pay-policy`. The inheritance is defined by the first three statements. The conditions for granting privileges are specifed by the remaining statements.

```
Role ExecMgr inherits InventoryMgr, AccountMgr.
Role InventoryMgr inherits Employee.
Role AccountMgr inherits Employee.

Permit(Employee, MakeRequest)
    := "request-box = None".
Permit(InventoryMgr, WriteOrder)
    := "request-box != None & (order-type = None)".
Permit(InventoryMgr, RejectRequest)
    := "request-box != None & (order-type = None)".
Permit(AccountMgr, ApproveOrder)
    := "(request-box != None) & (order-type != None) & (order-approval = 0)".
Permit(AccountMgr, RejectOrder)
    := "(request-box != None) & (order-type != None) & (order-approval = 0)".
Permit(ExecMgr, SubmitOrder)
    := "(request-box != None) & (order-type != None)
                                & (order-approval = 1) & (order-done = 0)".
```

The conditions follow the rules described in the previous description of the requirements for the system.

Adding Policy

Our tool is a Python script called `joinPolicy.py`. It is invoked with the previous two inputs using the following command:

```
python joinPolicy.py pay.smv pay-policy merged.smv
```

The first argument is the file name of the NuSMV model, the second argument is the policy, and the third is the file name for the merged output NuSMV model. It is the merged model which is used for verification.

NuSMV Results

The merged model's specification, which comes from the original NuSMV model, can be verified using NuSMV using the command `NuSMV merged.smv`. When the previous policy is used to produce the merged model, the specification should fail, giving the following counter-example:

```
-- specification  G (order-writer != order-approver | order-writer = None)
        is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  request-box = None
  order-type = None
  order-approval = 0
  order-done = 0
  order-writer = None
  order-approver = None
  Permit_Employee_RejectRequest = 0
  Permit_InventoryMgr_WriteOrder = 0
...
  Permit_AccountMgr_RejectOrder = 0
  Permit_ExecMgr_RejectOrder = 0
-> Input: 1.2 <-
  Role = AccountMgr
  Action = MakeRequest
-> State: 1.2 <-
  request-box = 2
  Permit_InventoryMgr_WriteOrder = 1
  Permit_ExecMgr_RejectRequest = 1
...
  Permit_ExecMgr_WriteOrder = 1
  Permit_InventoryMgr_MakeRequest = 0
-> Input: 1.3 <-
  Role = ExecMgr
  Action = WriteOrder
-> State: 1.3 <-
  order-type = 2
  order-writer = ExecMgr
```

```
  Permit_InventoryMgr_WriteOrder = 0
  Permit_ExecMgr_ApproveOrder = 1
...
  Permit_AccountMgr_RejectOrder = 1
  Permit_ExecMgr_RejectOrder = 1
-> Input: 1.4 <-
  Role = ExecMgr
  Action = ApproveOrder
-> State: 1.4 <-
  order-approval = 1
  order-approver = ExecMgr
  Permit_ExecMgr_ApproveOrder = 0
  Permit_AccountMgr_ApproveOrder = 0
...
```

Now, the counter-example shows the changing atomic propositions for all of the privileges. Note that the specification is violated a state 1.4, since `order-approver` and `order-writer` both equal `ExecMgr`. This is the state that violates the specification. Also note that `Permit_ExecMgr_ApproveOrder` is true at state 1.3, meaning that the executive manager is allowed to approve the order after approving it. This violates the specification.

6.2.3 Bug Resolution

In order to make the policy satisfy the separation of duties constraint, the policy must be modified. The policy has a problem when the `ExecMgr` attempts to invoke privileges inherited from both `InventoryMgr` and `AccountMgr` for the same order. This can be avoided by rewriting the rule for the action `ApproveOrder`:

```
Permit(AccountMgr, ApproveOrder) := "(request-box != None)
                                    & (order-type != None)
                                    & (order-approval = 0)
                                    & (Role != order-writer)".
```

Note the last condition added to the rule `(Role != order-writer)`. This should fix the rule due to the fact that the role of the order writer cannot also approve the order. Even though the rule specifies the condition needed for `AccountMgr`, it actually only affects `ExecMgr`. This is because of the inheritance: the privileges of `ExecMgr` must be a superset of `AccountMgr`.

Repeating the steps with the new policy, running NuSMV shows that the merged model satisfies the specification.

```
-- specification  G (order-writer != order-approver | order-writer = None)
      is true
```

Chapter 7

Protection in Dynamic Systems

The research in the field of access control presented by Harrison, Ruzzo, and Ullman [7] offers a model of a generally applicable protection system and then proposes decidability results concerning it. Their model has been used as a general basis for the field of access control and as such, pertains greatly to our project. Additionally, the decidability results which are proposed in their paper are very relevant to our project and must be considered when dealing with a protection system.

7.1 Modeling Protection Systems

The field of access control concerns the relationships between a set of objects and a set of subjects, which is contained as a subset of the set of objects. There also exists an access matrix P, which lists the subjects along the rows and the objects along the columns. For every subject and object there exists an intersection in P which contains a set of generic rights, or privileges, which a subject is allowed to perform on an object.

Definition. A *protection system* consists of a set R of generic rights and a set C of possible commands. The set R of generic rights contains all of the possible rights which can appear in the system.

A command in C consists of formal parameters which are used to specify locations in the access matrix, generic rights, and primitive operations on P. The formal parameters in a command are replaced by subjects and objects once the command is instantiated. The

generic rights are elements of R and will be searched for in the access matrix. If the rights are not found, then the command has no effect. If the rights are found, then the specified primitive operations are performed on the access matrix in the order they are given.

Definition. Definition: Given a subject s, an object o, and a generic right r, a primitive operation can be one of the following changes to the access matrix:

- enter r into P[s, o]
- delete r from P[s, o]
- create subject s'
- create object o'
- delete subject s
- delete subject o

where s', o' are formal parameters specified in a command c from C.

A primitive operation acts as a transition between configurations. In the paper it is proposed that these six primitive operations can be combined and duplicated to produce results which are equivalent to any possible modification which another operation might make to the access matrix.

The first primitive operation, enter r into $P[s, o]$, maintains the set S of subjects and the set O of objects. The operation maintains the access matrix as it previously was except at the location (s, o) where it adds the generic right r to the set of permissions. An important remark is that if the subject already has the right on the object then adding the right will not change the set of permissions.

The second primitive operation, delete r from P[s, o], also maintains the set S of subjects and the set O of objects. The operation once again maintains the access matrix as it previously was except at the location (s, o) where it removes the generic right r from the set of permissions. Once again, if the generic right was not in the set of permissions, then it cannot be removed and the primitive operation will have no effect on the access matrix.

The third and fifth primitive operations affect the set of subjects, the set of objects, and the access matrix. In the third operation, create subject s', the new subject is added to both the set S of subjects and the set O of objects, since S is a subset of O. Additionally, a new row and a new column are added to the access matrix with null sets at every intersection involving the new row and the new column. In the case of the fifth operation, delete subject s, the subject is removed from both S and O if it is already contained in those sets, and the row and column which correspond to the subject are also removed from the access matrix.

The fourth and sixth primitive operations are similar to the previous two mentioned. The fourth operation adds a new element o' to the set of objects, and adds a new column in the access matrix P with null sets where subjects interest with the new column in the access matrix. The sixth primitive operation, delete object o, the object is removed from the set of objects and the column corresponding to the object in the access matrix is also removed.

Definition. A configuration Q of a protection system is a triple (S, O, P) where S is the set of current subjects, O is the set of current objects, and P is a specific configuration of the access matrix.

Remark. A configuration Q is said to yield another configuration Q' if there is a command α whose parameters are found in Q, and if Q contains the necessary rights specified in α, and the sequence of primitive operations performed on the triple specified by Q results in the triple specified by Q'.

7.2 Transfer of Rights

Through the use of the model previously specified it is possible to describe various protection systems and the transitions between them. The important questions regarding access control do not surface with descriptions of the transitions, but rather with the analysis of the changes which occur to the access matrix as a component of those transitions, and what results that might have on the resulting environment the system is being applied to. Harrison, Russo, and Ullman discuss this through the context of the "safety" problem [7].

Definition. Given the context of a protection system, a command is said to *leak* a generic

right r from a configuration if at any point the command enters a generic right into the access matrix where the right did not previously exist.

It is not necessary that the right remain in the access matrix at the completion of the execution of the command since it is possible that during the execution of the command the access matrix was consulted and the rights were given to a user before the command finished executing.

Definition. Given a protection system, a configuration Q is said to be safe for a generic right r if there is no configuration Q' which Q yields such that a command α leaks r from Q'. If Q yields a configuration Q' which satisfies a command which leaks r from Q', then Q is said to be unsafe for r, or Q leaks r.

Now that there are terms which can be used to describe the transfer of rights, it is possible to analyze more detailed problems regarding access control.

Theorem. *1. For a protection system in which commands contain only one primitive operation, it is possible to develop an algorithm which will always determine whether or not an initial configuration is unsafe for a given generic right.*

Proof. Since the only commands being considered are commands which contain a single primitive operation, commands will be referred to by reference to the operation which they perform.

The basis for the first theorem lies in two restrictions to the capabilities of the model. The first restriction is one which is also exhibited in Datalog, in that the absence of a privilege or an object cannot be tested by commands. A command can only check if the right or object is present; it takes no action if it cannot find the appropriate values. Due to this restriction one only needs to consider three of the primitive operations, the enter operation and the two creation operations. Since the absence of a right cannot be determined by a command, there cannot be a command which requires a delete or destroy operation to occur before the command in order for it to leak a right. For any configuration Q', with a configuration Q which yields Q' through either a delete or destroy command, if Q' is unsafe for a generic right r then Q is also unsafe for r, since none of the commands which lead to a leak of r can

check for the absence of the objects or rights removed by the delete or destroy command executed on Q in order to yield Q'.

The second restriction to the model which allows for the first theorem pertains to a limit on the length of a minimal sequence of commands which leads to the leak of a right. The restriction here lies in commands within the system only being allowed to have one primitive operation.

In addition to the ability to ignore delete and destroy commands in the context of a leak, there is a restriction on the number of create commands which can appear in the computation, the sequence of commands, which leads to a leak of generic right r. In the paper it is proposed that there will be at most one non-enter command in the shortest computation which leaks a right.

Any destroy or delete commands, as specified before, can be removed from the computation. As a result, the only possible non-enter commands which can exist in a shortest computation which leaks a generic right r are either create subject commands or create object commands.

Consider the case of a create subject command c in a sequence of commands where c is the last create command. Assume the create subject command takes effect on a configuration where there were no subjects previously. There can be no rights specified in the access matrix since there will have been no subjects to which rights would be granted. Consequently, all previous commands can be ignored and removed from the computational sequence since their operations had no result on the access matrix. Finally, the objects which exist in the configuration can be merged with the subject s created in the command c. Any further commands which refer to one of the previous objects now refer to the s. As a result of these steps, a shorter computation has been formed which leaks the right r from the original configuration.

Consider a create object command which takes place in a computation as the last create command in the sequence of commands in a computation. The set of subjects remains the same after the execution of the command as it was before the execution of the command. Since there has to be at least one subject in a configuration in order to leak a right, and the create object command does not alter the number of subjects, than the set of subjects

must contain at least one subject before the execution of the command. For reference, refer to the object created by the create object command as o. Additionally, since the previous paragraph considered the case of a create subject command being executed on a configuration where the set of subjects started empty, the case of a create subject command with the set of subjects initially containing at least one subject must be considered. Since all subjects are also considered objects, then, without loss of generality, the subject created by the create subject command can be referred to as o as well, and both commands can be referred to as a create command c. Since, for either case of c, the set of subjects is not empty, consider at least one subject s in the initial set of subjects. By replacing all instances of o in all further enter commands, which are the only commands remaining after c by the assumption made, the create command c can be removed without changing the property that the computation in question leaks the given generic right. Any references to the rights which the object o, which was instead not created in the shorter computation, will instead refer to the rights which the subject s possesses, allowing all further enter commands to occur as they previously would have before the removal of the command c. This result can be repeated for any create command which occurs on a configuration with a non-empty set of subjects.

Finally, since the set of generic rights is finite, the set of current objects has to be finite, and there can be at most one create command, then the number of single-operation enter commands which have a unique effect on the access matrix P must also be finite. Since there is a limit on the length of the shortest computation since any commands which would duplicate the effect of another could be removed the algorithm will always complete. □

Definition. A Turing machine consists of a finite set of states K, a finite set of tape symbols G and a finite set of transition functions. The set of states contain a starting state, an accepting state and a rejecting state.

The set of tape symbols is a set of characters which can appear on the input tape for the Turing machine. The input tape starts at an initial point and continues infinitely. Each symbol on the tape is an element of the set of tape symbols, which contains a blank symbol b. The Turing machine also contains a tape head to keep track of the current location on the tape.

A transition function in a Turing machine is a mapping $l : K \times G \rightarrow K' \times G' \times \{L, R\}$ from a state and a symbol to a state, a symbol and a direction to move on the tape, either left or right. The state K refers to the current state that the Turing machine exists in, and the symbol G consists of the current symbol that the tape head is at. The state K' is the new state that the Turing machine transitions to and the symbol G' is the new symbol which replaces the symbol under the tape head. As mentioned, the tape head then moves one step in the direction specified in the transition function.

There are a large number of questions which can be simplified to a true or false question or a series of true or false questions. One such question in computer science is whether a Turing machine and an input string given to the Turing machine will eventually reach a halting state or if the Turing machine will continue indefinitely.

Definition. It is said that a true or false problem is *recursively enumerable* if there exists a Turing machine to solve the problem which will always end in an accepting state if the problem is true.

One important consideration about a recursively enumerable problem is that the Turing machine might not end in a rejecting state if the problem results as false.

Definition. A true or false problem is said to be decidable if the language is recursively enumerable and the Turing machine will always reach a halting state regardless of whether the result of the problem is true or false.

Remark. A significant problem in computer science is the halting problem. It asks the question of whether a given program with given input will loop forever or if the program given will eventually halt. This problem is known to be undecidable. It can be shown that the problem is undecidable by first considering a program H which outputs a result of "halt" if the program and input given to it will halt, and H outputs "loop" if the program and input given to it will loop indefinitely. Consider another program K which will take the output of H as its input and output "halt" if H outputs "loop", and will loop forever if H outputs "halt". If the program K is given itself as input, meaning that H takes K as input and the result of H will be given to K, then either H will output "halt" which will cause K to loop forever, or H will output "loop" which will cause K to halt. Either result will prove

that H does not answer the question for any program. Since the method that H uses to discern whether the program given it halts or not has been unspecified, this result holds for any possible H, thus showing that H cannot exist and the halting problem is undecidable.

Remark. One important feature of a protection system which is clarified in [7] is that a protection system can be used to represent a Turing machine.

To begin, we state that the generic rights which exist in a protection system contains the set of tape symbols, the set of states of the Turing machine, and the additional rights own and end. The commands in a protection system are used to represent the transition functions of the Turing machine. Note that in this case the commands can possess more then one primitive operation. The tape for the Turing machine, which contains a sequence of cells in which a single symbol can be held, is simulated through the access matrix in the protection system. The set of objects is equal to the set of subjects in the representation which are ordered in an arbitrary manner. Each subject s_i has the generic right 'own' to the next ordered subject s_{i+1} in the access matrix. The final subject in the ordering possesses the generic right 'end' to itself. Additionally, starting with the first subject s_1 and the first cell in the access matrix and continuing incrementally, the subject is given the right to itself which corresponds to the symbol in the appropriate tape cell. As such, if the ith tape cell has the symbol X, then the subject si will have generic right X to itself. To complete this model, if the tape head in the Turing machine is at cell i, then the subject s_i posses a generic right to itself corresponding to the state that the Turing machine is currently in. The initial set of subjects consists only of those subjects which correspond to the cells of the tape for the Turing machine which do not contain a blank value. There will be a command in the protection system model of a Turing machine for each possible transition, which takes two subjects as parameters. The command will check that the first state has the 'own' right to the second state. Then it will check for the presence of the generic rights corresponding to the state and the symbol that the transition requires as parameters. If present, the command will replace the tape symbol at the first state with the new tape symbol specified by the transition, remove the current state symbol, and place the new state symbol in the second state's location in the access matrix. There are additional commands which copy the previous set of commands and, when encountering the generic right end, will mimic

their counterparts except they will additionally add the new subject before indicating the movement. As a result, the only commands which will be allowed will be the commands which originate from the current state in the Turing machine.

Theorem. *2. Determining if a given configuration is safe for a given generic right within a protection system, which will be referred to as the safety question, is undecidable.*

Proof. To prove this, it is sufficient to show that deciding if a Turing machine halts can be reduced to determining the decidability of the safety question. This condition of expressing the solution to one Turing machine problem in terms of another problem is referred to as Turing-reducibility. To show that the safety problem is undecidable, one only needs to show that a solution to the safety problem gives a result for the halting problem. Since the halting problem is undecidable, no such result can exist and there can be no decidable result for the safety problem, making the safety undecidable. To this extent, it has been shown that a protection system can mimic the components and actions of a Turing machine, allowing it to perform the same functionality of any Turing machine. This gives a direct correlation between any results about protection systems and Turing machines. The halting problem, which is the question of whether or not a Turing machine will reach a halting state, can be related to the problem of safety, which asks whether an initial configuration is unsafe for a generic right r, by relating the right r to a halting state. If the Turing machine represented by the protection system reaches a halting state, then the protection system will leak the generic right which corresponds to the halting state.

Although the two theories might seem similar upon initial inspection, there are significant distinctions between them. The first theorem is asking the question of a given system being unsafe for a generic right, while the second theorem is asking the question of a given system being safe for a generic right. For the first theorem, the authors propose that for any given system there is an algorithm which allows for the discovery of the computation which leaks a given right r from an initial configuration Q. The algorithm which is proposed will run on the protection system starting at Q until it discovers the shortest computational sequence which leaks the right r. Since the shortest computational sequence is bounded if Q is indeed unsafe for r, then the time which the algorithm will take to discover the shortest computational sequence is also bounded. Additionally, this leads from the fact that the set of generic

rights and the set of current objects are finite, forcing the number of possible computations consisting of single-operation commands to be finite. Note that since computations involving multiple create commands can be ignored, the set of current objects can be considered finite throughout any given computational path. □

7.3 Results

The paper presented by Dougherty, Fisler, and Krishnamurthi [5] presents a means of modeling an access control policy and the system which the policy acts upon. Within their paper they propose a theorem, to be referred to as theorem 3, that determining if a system can reach a state which will satisfy a policy statement is decidable.

The three theorems specified thus far discuss similar events. Theorem 1 discusses access control possessing the trait of being recursively enumerable. It does not conjecture that determining if a configuration within a protection system leaks a right is decidable; it only states that the affirmative case of the problem can be determined. Consequently, theorem 1 does not conflict with either of the remaining two theories.

Both the second and third theories ask questions about access rights being granted within their respective systems, and both theories specify distinct results about the decidability of these questions. The second theorem introduces the policy as a protection system and equates that to a Turing machine. It then considers the case of any real-world system acting in accordance with the policy executing a sequence of commands which will lead to a specific right being granted to a subject. Since the policy is acting upon any real-world system it is unclear what effect the policy has on the system and it is unknown when various commands might be made to affect the policy.

The third theorem introduces a policy which acts upon a specified system. The system given is defined as a finite state automaton which mimics a real world system. The policy is altered based upon the current state of the system, and when it is applied to the system it restricts the transitions which are available from various states. In this manner the effect which the policy and system have on each other is clear.

The distinction between the two theories lies in the both specification of a system along

with the policy by the third theorem, and that the system specified by the third theorem is a finite state automaton. Since there is no specific system specified by the second theorem, reasoning about its policy must be done about any possible system it works in conjunction with. Additionally, the system which the second theory might hypothesize about might not be deterministic, or contain a finite number of states it can exist upon.

Chapter 8

Conclusions & Future Work

We have demonstrated that model checking can be used to verify access-control policies in a dynamic environment. The result is a tool which is backed by mature work done in model checking, and also allows great flexibility in describing environments.

One future extension of our work would be to be able to model user sessions and user assignments to roles in the policy or the model. Our design does not take into account the specific users who are invoking roles to perform actions during a session of the abstracted system. Remember that users are granted privileges indirectly through roles. It would be useful to policy checking to be able to take into account the assignments of roles to users. For example, if users can be handled, then specifications could be written that check how users behave. This would be a big improvement since only the roles which are invoked can be refered to.

For example, imagine a homework submission/grading program. Students can register for course, submit homework, and review grades. On the other hand, professors and teaching assistants write assignments and grade user submissions. Certain things should be expected in such a system: no student can be both registered for a class and be assigned as a teaching assistant by the academic department. This constraint cannot be modelled with our current tool since we only specify the existing roles and not the users in roles.

Ultimately, there are limitations with both NuSMV and RBAC. NuSMV can only handle propositional logic, meaning it can't handle full Datalog policies. RBAC is also propositional in nature because it cannot be used to reason directly about users, such as in a UNIX file

system. Role-based access control is not the ultimate goal of our work. More work could be performed to produce a similar tool to verify model with policies expressed in more complex access-control paradigms, such as lattice-based access control [11] which is more general than RBAC and is used to write more complex policies. Work with more advanced paradigms will require lifting our work into a more powerful form in the future.

Bibliography

[1] Serge Abiteboul and Richard Hull. Data functions, datalog and negation. In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 143–153, New York, NY, USA, 1988. ACM Press.

[2] Roberto Cavada Alessandro. Nusmv 2.4 user manual.

[3] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking, 2002.

[4] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[5] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *IJCAR*, pages 632–646, 2006.

[6] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 196–205, 2005.

[7] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. On protection in operating systems. In *SOSP '75: Proceedings of the fifth ACM symposium on Operating systems principles*, pages 14–24, New York, NY, USA, 1975. ACM Press.

[8] Michael Huth and Mark Ryan. *Logic in Computer Science*. Cambridge University Press, second edition edition, 2005.

[9] Butler Lampson. Protection. In *Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems*, pages 437–443, Princeton University, 1971.

[10] Ammar Masood, Arif Ghafoor, and Aditya Mathur. Scalable and effective test generation for access control systems that employ rbac policies. Technical report. CERIAS TR 2006-24.

[11] Ravi S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, 1993.

[12] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[13] Ravi S. Sandhu and Pierrangela Samarati. Access control: Principles and practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.